PipeCNN：一种基于软件流水线的并行化卷积神经网络方法 *

吴　鹏†，周宁宁

(南京邮电大学 计算机学院，江苏省 南京 210023)

摘　要：针对使用传统的模型并行方法加速卷积神经网络训练容易出现设备利用率不高的问题，提出了通过软件流水线方式加速卷积神经网络的方法 PipeCNN。首先研究了卷积神经网络中的前向传播与反向传播算法，分析了训练过程中的数据相关性，然后基于软件流水线改进了卷积神经网络的训练过程，提出了两种可行的参数更新方式，最后使用循环队列来实现网络中层与层之间的消息通信，并提出任务分配算法来划分软件流水线中的工作段。实验结果显示，这种方法在保证模型准确性的前提下，可以取得良好的加速比以及设备利用率，表明了使用软件流水线可以有效解决模型并行中设备利用率不高的问题，提高了卷积神经网络的训练速度。

关键词：卷积神经网络；软件流水线；模型并行；深度学习

中图分类号：TP391　　doi: 10.19734/j.issn.1001-3695.2020.02.0038

PipeCNN: parallelization of convolutional neural network based on software pipeline technology

Wu Peng†, Zhou Ningning

(*School of Computer*, *Nanjing University of Posts & Telecommunications*, *Nanjing* 210023, *China*)

Abstract: Aiming at the traditional models parallel method for accelerating convolution neural network (CNN) tend to have low utilization, this paper proposed PipeCNN, a method that accelerates CNN with software pipeline. Firstly, this paper studied the forward propagation and back propagation, and then explored data correlation during training. Secondly, it parallelized CNN with the support of software pipeline, and then analyzed two feasible gradient updating methods in PipeCNN. Finally, it used circular queue to realize communication between two layers and then proposed a task allocation algorithm to divide CNN into working parts. Experiments showed that the method can obtain good speedup and utilization while ensuring the accuracy of the model. It showed that software pipeline can effectively solve the problem of low utilization in model parallel, and accelerate the training of CNN.

Key words: convolutional neural network; software pipeline; model parallelism; deep learning

0　引言

卷积神经网络是深度学习中最具有代表性的一种模型，典型的卷积神经网络是由一定个数的基本结构单元反复堆叠而成，近些年来，这种网络在结构上呈现出逐渐加深的趋势，改善了网络的特征提取能力[1]。Resnet[2]通过越层连接的方式很好地解决了网络层数增多带来的梯度消失问题，并开创性地将卷积神经网络的层数增加到 200 层甚至上千层，提高了模型在数据集上的表现能力。扩展模型深度意味着增加模型参数以及计算量，在实际的应用中，单台设备越来越难以满足存储并计算大量数据的需求，因此使用并行技术[3~6]来加速深度学习渐渐成为主流的做法。

流水线[7]是一种常用的并行加速技术，其适用于具有循环体结构的任务。文献[8]使用流水线来缓解 CPU 与 GPU 通信开销带来的计算效率的问题。文献[9]提出一种适用于 X86 平台的软件流水实现框架，用流水线实现了处理器的部分硬件行为。文献[10]设计了一种软件流水循环缓冲，用于存储和派发循环体指令，从而减少访存延迟对性能的影响。文献[11]基于多核处理器提出了一种面向数据流程序的软件流水并行化方法,达到最大化流水线计算速率的同时最小化通信开销的目的。

流水线在深度学习中也得到了越来越广泛的应用，为了满足高效的模型并行需求，文献[12]提出了 Gpipe，这是一种基于流水线并行加速深度学习的方法，其核心思想是使用分批的流水算法，将模型分配于多个加速设备中，从而可以达到线性的加速比。但是在模型的训练过程中，Gpipe 不可避免地会出现空闲阶段，这会限制设备利用率的提升。文献[13]为了解决传统 GEMM 卷积算法在神威太湖之光超级计算机上执行效率不高的问题，使用软件流水的方法降低从核访问的开销，使得并行卷积算法比基础算法有了 90 倍的性能提升。

在深度学习中，卷积神经网络的堆叠式的结构比较符合软件流水线中循环体的特征，因此软件流水技术可以很好地应用在并行加速卷积神经网络的任务中。Gpipe 利用了软件流水的思想加速了深度学习的计算，却容易出现空闲阶段限制设备利用率的问题，因此可以在此基础上提出一种改进方案，达到消除流水线空闲阶段的效果。

1　前向传播与反向传播

实现基于软件流水线的卷积神经网络并行计算方法 PipeCNN，需要先探究前向传播与反向传播过程中的数据依赖关系，从而保证数据的相关性以及并行计算的正确性。

1.1　前向传播

在前向传播中，卷积神经网络接收输入数据，并从前往后逐层进行神经元的计算，最后输出计算结果。使用 w^l 表示第 l 层中的权值，b^l 表示第 l 层的偏置，a^l 表示 l 层中神经元的输出，并用符号 $\sigma(v)$ 代表激活函数的计算，符号 $*$ 代表卷

收稿日期：2020-02-14；修回日期：2020-05-13　基金项目：智能电网保护和运行控制国家重点实验室开放课题(201610，20169)；国家自然基金资助项目(61170322，61373065，61302157)

作者简介：吴鹏(1995-)(通信作者)，男，江苏连云港人，硕士研究生，主要研究方向为深度学习(1055408601@qq.com)；周宁宁(1972-)，女，江苏南京人，副教授，硕导，博士，主要研究方向为计算机视觉.

积运算，则第 l 层中神经元输出的计算过程如式(1)所示。

$$a^l = \sigma(w^l * a^{l-1} + b^l) \tag{1}$$

卷积神经网络的前向传播是一个循环迭代的过程，式(1)表示了第 l 层的前向计算，然后，需要将第 l 层的计算结果作为输入，并与当前层的参数进行卷积运算，才能得到 l+1层的输出。卷积神经网络前向传播的计算流程如下：网络中的输入数据首先传送给 a^0，然后从首层开始，逐层执行前向计算函数，到最后一层即可得到输出结果。

1.2　反向传播

在训练过程中，前向传播可以持续计算直到产生一个标量代价函数,反向传播将来自代价函数的信息向后流动，计算出梯度数据。反向传播算法的目的是计算损失函数的 C 关于权重 w 以及偏置 b 的偏导数$\partial C/\partial w$ 以及$\partial C/\partial b$，主要有以下 4 个步骤：

a) 前向传播完成后，计算来自网络最后一层，即第 L 层的误差，如式(2)所示。其中 $\nabla_a C$ 表示损失函数对第 L 层的输出产生的偏导数，符号 ⊙ 表示 Hadamard 积，该操作可以返回两个相同维度的向量逐元素相乘之后的结果。

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2}$$

b) 从后往前依次计算出每一层的误差 δ^l，得到式(3)。由于使用式(2)可以计算出 δ^L，使用式(3)可以计算出 δ^{L-1}，因此可以类推下去，计算出 δ^{L-2}，δ^{L-3} 等等，从而得到所有的 δ^l。

$$\delta_j^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3}$$

c) 计算损失函数关于权重偏置 b 的偏导数 $\partial C/\partial b_j^l$，得到：

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{4}$$

d) 计算出网络中所有层权重 w 的偏导数 $\partial C/\partial w_{jk}^l$，得到：

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{5}$$

计算完偏导数后，卷积神经网络需要采用合适的优化算法来更新参数。现在一般采用小批量梯度下降方法[14]对深度学习进行优化，这种方法是使用一个批次中数据的分布来近似整个数据集的分布情况，通过一批数据得到的梯度可以用来更新参数，同一批的数据共同决定了本次梯度的方向。

1.3　反向传播算法中的数据相关性

图 1 所示为前向传播与反向传播的大致过程，可以清楚地观察到数据流动的方向：在前向传播中，数据逐层从前往后流动，最后一层计算完后产生标量代价函数，然后数据逐层从后往前流动，并得到每一层的梯度数据，当数据再次流动到首层时，反向传播算法结束。

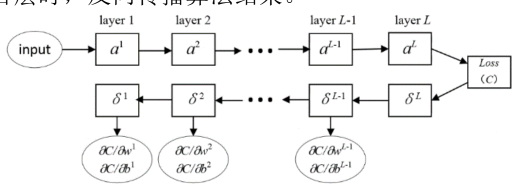


图 1　数据传播过程

Fig. 1　Data currency

为了研究传播过程中的数据相关性，使用伪代码的形式来表示卷积神经网络的反向传播算法，如算法 1 所示。

算法 1　反向传播算法

```
输入：Input data x
输出：∂C/∂w^l_jk and ∂C/∂b^l_j
a^1 ←x
for l = 2 to L do
    z^l← w^l * a^(l-1)+b^l
    a^l ←σ(z^l)
end for
δ^L ←∇_a C ⊙ σ'(z^L)
for l = L-1 to 1 do
    δ^l_j ← ((w^(l+1))^T δ^(l+1)) ⊙ σ'(z^l)
    ∂C/∂w^l_jk ← a^(l-1)_k δ^l_j
    ∂C/∂b^l_j ← δ^l_j
end for
```

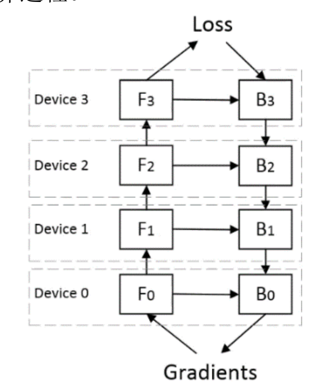
从算法 1 可以分析出数据之间的依赖关系：在前向传播中，从卷积神经网络的第一层开始，依次逐层执行前向计算的函数，其中第 l 层 a^l 的计算需要依赖于第 l-1 层输出的结果 a^{l-1}；而在反向传播中，数据从后往前流动，其中第 l 层 δ^l 的计算需要依赖于第 l+1 层计算出的 δ^{l+1}，而偏导数 $\partial C/\partial w_{jk}^l$ 的计算又依赖于 δ^l 和第 l-1 层的神经元输出结果 a_k^{l-1}。

2　基于流水线的卷积神经网络并行化方法

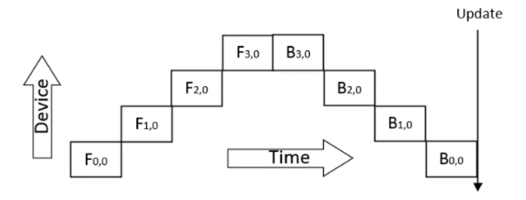
2.1　卷积神经网络中的并行计算方法

在卷积神经网络的应用中，有两种并行计算的方法，分别为数据并行和模型并行。数据并行是一种较常用的模型加速方法，这种方式需要同时在多个计算设备中使用相同的模型副本，并将一个批次的数据分配在不同的计算设备中进行前向传播和反向传播的计算，然后使用合适的优化算法更新模型参数。神经网络使用更新过后的参数继续进行下一批次的训练，并按照相同的方式迭代下去。

当模型的规模较大，而单个设备无法存放整个模型的时候，会采用模型并行的方法来实现训练过程。将一个设备中执行的计算任务称为工作段，则这种方式需要将模型分为 d 个较小的工作段，并放入不同的设备中执行，从而可以通过多个计算设备实现大规模的模型训练任务。图 2 是 d 为 4 时的模型并行计算过程。



(a)模型并行中的前向传播与反向传播



(b)模型并行的时空图

图 2　模型并行

Fig. 2　Model parallelism

从图 2(a)可以看出模型并行中前向传播与反向传播的计算过程，F_k 表示该工作段在第 k 个设备中执行前向传播计算，B_k 表示该工作段在第 k 个设备中执行反向传播计算。计算过程中的数据依赖关系通过箭头表示出来，其中工作段 F_{k+1} 的计算依赖于 F_k 中的数据，而工作段 B_k 的计算需要同时依赖于 B_{k+1} 以及 F_k 中的数据。图 2(b)是模型并行的时空图，其中 $F_{k,j}$ 表示第 k 个设备执行第 j 条数据的部分前向计算，$B_{k,j}$ 表示第 k 个设备执行第 j 条数据的部分反向计算。由于存在数

据相关性的问题，同一时刻只有一个设备在参与计算，而其他的设备需要等到相关的设备计算完毕才能运行。

2.2　PipeCNN 的模型并行

软件流水线是一种常用的并行加速手段，而将这种技术应用于深度学习中具有很大的实用价值。文献[12]提出了 Gpipe，这是一种使用流水线实现模型并行的框架，缓解了模型并行中设备利用率不高的问题。该方法将一个批次的训练样本进一步划分为多个微批次，不同的计算设备可以同时执行不同微批次中的数据，当一个批次中所有微批次的数据计算完毕后，再执行同步梯度更新算法来改变权值。图 3 是 GPipe 中微批次的数据个数为 1 时，流水线的时空图，可以看出前向传播到反向传播的过渡阶段仍然会出现空闲的时间，这也限制了设备利用率的进一步提高。

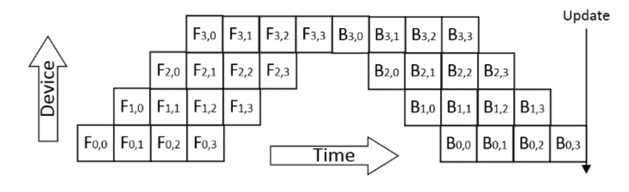


图 3　Gpipe 中的时空图

Fig. 3　Space-time diagram in Gpipe

为了解决 Gpipe 中的流水线在计算过程中会出现的空闲等待的问题，提出一种新的加速卷积神经网络的方法 PipeCNN。虽然 PipeCNN 也是基于流水线的思想，但与 Gpipe 不同的是，其中的前向传播和反向传播的计算可以灵活地分配在不同设备中执行。例如，图 4 中的 PipeCNN 将前向与反向的计算分配在了不同的工作段中，从而将计算设备增加至 8 个。这种方式相当于进一步细分了卷积神经网络中的计算任务，消除了前向与反向的计算必须绑定在同一个设备中的限制，增加了工作段中任务分配的灵活性。

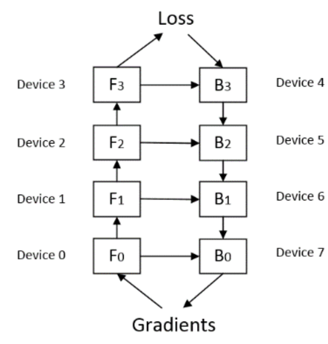


图 4　PipeCNN 中的模型并行

Fig. 4　Model parallelism in pipecnn

2.3　PipeCNN 的权值更新

PipeCNN 加快了卷积神经网络中梯度的计算过程，然而，如何在使用流水线计算梯度的同时更新权值是一个重要的问题。理想的情况下，权值更新的过程与 PipeCNN 计算梯度的过程可以正常执行，互不干扰，然而在实际的操作中会出现两种情况：

a) 如果 PipeCNN 按照同步梯度更新的方式进行计算，那么当流水线计算完一个批次之后，必须停止工作，等到权值更新完毕之后，才能够让流水线重新运行，进行下一个批次的计算，将这种方式命名为间断性 PipeCNN。从图 5 的时空图可以看出，设备利用的效率和批量的大小有关，当批量较小的时候，设备的空闲时间较长，导致利用效率低，而批量增大的时候，设备利用效率会逐渐提高。

b) 如果保证 PipeCNN 梯度的计算连续进行，就需要在当前批次数据的梯度还未计算完毕，就把下一个批次的数据送到网络中进行计算，这会导致某些训练数据的前向计算过程中使用参数版本不一致的现象，继而影响后续批次的梯度计算过程。例如，对于第 j 个样本，其前向传播中的前部分使用的是第 i 个版本的权值，而后部分使用的是第 i+1 个版本的权值，然后将模型最后的输出结果用于梯度的计算。这种方式会使得一部分的数据不能在训练过程中使用完整的新参数进行前向计算，在一定程度上减弱了新参数的作用。这实际上是一种异步梯度更新[15]的方式，这种方式可以使模型收敛，但是可能会导致模型训练在训练过程中出现不稳定现象，将这种方式命名为连续型 PipeCNN。图 6 是批量为 6 时连续型 PipeCNN 的参数更新情况。

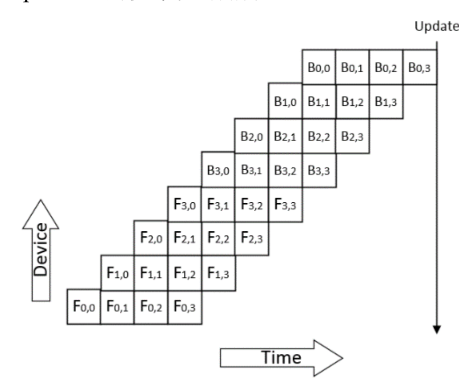


图 5　间断型 PipeCNN

Fig. 5　Interval pipecnn

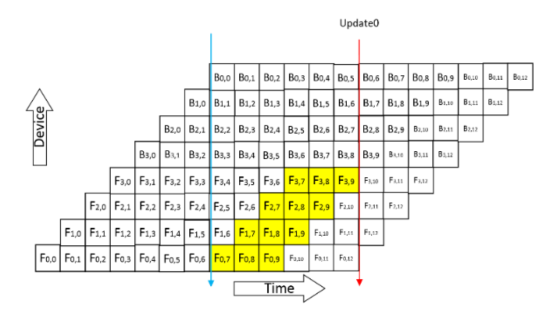


图 6 连续型 PipeCNN

Fig. 6　Consecutive pipecnn

比较 PipeCNN 中两种参数更新的方式可以发现，连续型 PipeCNN 消除了参数更新的空闲等待阶段，具有较大的优势，因此下面着重分析批量的大小对连续型 PipeCNN 的影响。观察图 5 中两个箭头中间的数据可以看出，第 7，8，9 条数据在前向传播的过程中使用了完整的同一版本的参数，而第 10,11,12 条数据在前向传播过程中会使用两个版本的参数。在连续型 PipeCNN 中，这种异步梯度更新的方式是不可避免的，但是增大批量的大小就会观察到，使用同一版本参数进行前向计算的数据增多，而使用不同版本参数的数据个数却不会发生改变，从而异步梯度更新的数据所占的比例会逐渐减少。

设前向传播连续型 PipeCNN 的设备数为 d，而批量的大小为 b，且 b>d，则在同一批次中，有(b-d+1)个样本使用了同步梯度更新的方式进行了计算，而只有(d-1)个样本使用了异步梯度更新的方式进行计算，因此一个批次样本中使用同步梯度更新的比例为

$$P_s = \frac{b-d+1}{b} \tag{6}$$

从式(6)可以看出，当 b>>d 的时候，P_s 接近于 1，此时大部分的数据都会使用同步梯度更新的方式，这种情况下网络的收敛是稳定的。在深度学习中，采用较大的批量进行网络训练是一种常见的做法，以便一个批次的数据可以更接近于整个数据集的分布情况，因此连续型 PipeCNN 可以应用到大部分的深度学习训练任务中。

3　PipeCNN 的具体实现

3.1　基于循环队列的消息通信机制

PipeCNN 可以将卷积神经网络中的不同层抽象为最小任务单元，多个任务单元可以组合成为工作段，因此需要解决任务单元之间的通信问题。由于实验主要在 CPU 的环境下进行，因此该问题可以很方便地使用多线程共享变量的方式实现。为了保证计算过程中数据顺序的正确性，使用循环队列来存储卷积神经网络每一层的数据。

循环队列是一种线性数据结构，它遵循先进先出的原则，队列中最后一个元素的 rear 指针指向第一个元素的位置，可以形成一个环。在定义循环队列的时候，需要指定队列存储数据的个数，并开辟相应的内存空间。循环队列中一共有三种方法，分别为 *Front()*，*enQueue()*，*deQueue()*，它们的作用分别是返回队头元素，入队和出队操作。一般来讲，在一个空队列上执行一个出队的操作会引发异常，为了保证在 PipeCNN 中使用循环队列这种数据结构而又不会发生异常，可以使用 while 循环进行等待，并一直执行判断直到相关操作可以执行便跳出 while 循环。这种方式不需要严格规定工作段之间的同步，同时也保证了计算的正确性。

前向传播中任务单元的执行过程如算法 2 所示，当前的任务单元可以读取前一个单元的输出数据作为输入，然后进行前向计算，处理完毕后，将计算结果放到循环队列中，等待下一个任务单元读取数据。

算法 2　PipeCNN 前向传播

输入：layer$^{(l)}$,layer$^{(l+1)}$

输出：$a^{(l+1)}$

a^{l} ←layer$^{(l)}$.Front()

z^{l+1}← w^{l+1}* a^{l}+b^{l+1}

a^{l+1} ←$\sigma(z^{l+1})$

layer$^{(l+1)}$.enQueue(a^{l+1})

layer$^{(l)}$.deQueue()

反向传播计算过程中用到的数据主要有 δ^{l}，z^{l}，w^{l}，在计算机中将它们表示为可以处理的张量形式。在 PipeCNN 中，δ^{l} 的数据存储在 layer 类中名为 prev_delta 的队列中，z^{l} 的数据存储在 layer 类中名为 output_z 的队列中，w^{l} 的数据存储在 layer 类中名为 weight 的队列中。PipeCNN 任务单元的反向传播如算法 3 所示。

算法 3　PipeCNN　反向传播

输入：layer$^{(l-1)}$,layer$^{(l)}$, layer$^{(l+1)}$

输出：$\partial C/\partial w_{jk}^{l}$，$\partial C/\partial b_{j}^{l}$

δ^{l+1} = layer$^{(l+1)}$.prev_delta.Front()

z^{l} = layer$^{(l-1)}$.output_z.Front()

w^{l+1} = layer$^{(l+1)}$.weight.Front()

δ^{l} = $((w^{l+1})^{T}\delta^{l+1})\odot\sigma'(z^{l})$

$\frac{\partial C}{\partial w_{jk}^{l}}=a_{k}^{l-1}\delta_{j}^{l}$

$\frac{\partial C}{\partial b_{j}^{l}}=\delta_{j}^{l}$

layer$^{(l)}$.prev_delta.enQueue(δ^{l})

layer$^{(l+1)}$.prev_delta.deQueue ()

layer$^{(l-1)}$.output_z. deQueue ()

layer$^{(l+1)}$.weight. deQueue ()

需要注意的是，由于前向传播中产生的数据可能要在若干个时刻之后才能在反向传播中用到，因此需要保证 PipeCNN 中队列的容量要尽量大，在实验中，设置队列的容量为 100。

3.2　工作段的划分

在软件流水线中，多个工作段的任务会分别交给不同的计算资源执行，为了取得良好的加速比，需要保证每个工作段的执行时间大致相同，因而需要有一个较为合理的任务分配算法。可以对任务分配的问题进行抽象：给定大小为 n 的任务单元执行时间的集合 W={w_1,w_2,…,w_n}，集合中元素按照固定的顺序排列，并给定工作段个数 d,其中 $n\geq d$，任务分配算法的目的则是将集合 W 划分为 d 个互不相交的子集 W_1,W_2…W_d,满足 $W_1\cup W_2\ldots\cup W_d=W$，并且这 d 个子集的任务执行时间都尽可能地相等，即时间 t_{avg}，如式(7)所示。

$$t_{avg}=(\sum_{i=1}^{n}w_i)/d \tag{7}$$

但在实际情况中，很难保证每个任务子集中的执行时间都相等，因此需要采取措施尽量缩小不同任务子集的时间差距。任务分配算法的基本思想如下：对集合 W 中任务单元的执行时间依次进行累加，取累加时间与 t_{avg} 最接近的任务组合为任务子集，并按照同样的方法，进行下一个任务子集的分配，直到集合 W 中的所有任务都分配完毕。具体如算法 4 所示，其中 T_{pre} 表示保存上一次时间累加结果，T_{cur} 表示当前的时间累加结果，L 表示任务子集左边界的位置，R 表示任务子集右边界的位置。函数 minDistance(T_{pre}, T_{cur}, t_{avg})返回 T_{pre} 和 T_{cur} 中更接近 t_{avg} 的任务单元编号。

算法 4　任务分配算法

输入：任务时间集合 W{ w_1,w_2,…,w_n }，工作段个数 d

输出：任务子集 W_1,W_2…W_d

$t_{avg}=(\sum_{i=1}^{n}w_i)/d$

T_{pre} = 0; T_{cur} = 0 ;L = 0;R=0;

for i = 1 **to** n **do**:

　T_{pre} = T_{cur}　　//保存上一次的累加结果

　　T_{cur} = T_{cur} + w_i　//得到当前的累加结果

　　if T_{cur} ==t_{avg} :

　　　R = i;

　　　添加任务子集{w_L..w_R};

　　　T_{pre} = 0; T_{cur} = 0 ;L = i+1;

　　if T_{cur} >t_{avg} :

　　　R = minDistance(T_{pre}, T_{cur}, t_{avg});

　　　添加任务子集{w_L..w_R};

　　　if $R=i-1$:

　　　　T_{cur} = T_{cur} - T_{pre};

　　　else:

　　　　T_{cur} = 0;

　　　T_{pre} = 0; L = R+1;

使用算法 4 可得到 d 个工作段的具体分配内容，在任务单元个数较多且任务粒度较小的情形下，该算法分配结果较好，因此算法 4 可以解决深度卷积网络中的任务分配问题。划分完成之后的工作段可以在多核设备中执行，使用多线程技术来实现并行计算，其中每个线程执行的计算与工作段的任务相对应，且相邻工作段的线程使用队列数据结构进行信息传递，PipeCNN 正是以这种思想来加速卷积神经网络的计算过程。

4　实验结果及分析

实验采用连续型 PipeCNN 的方式来研究其中并行计算的加速比以及设备利用率，主要在 MNIST 数据集以及 Cifar10 数据集上进行了并行计算的分析。

4.1　MNIST 数据集实验

首先进行训练识别 MNIST 手写字符的实验，该实验既

要保证模型训练的正确性，又要通过 PipeCNN 的并行计算方法达到加速度的目的。实验中根据 MNIST 数据集设计出具有 6 层的卷积神经网络，设置参数 Batch Size 为 32，Epoch 为 10，串行执行卷积神经网络可以得到训练过程中每一层的平均计算时间如表 1 所示。

表 1 每一层的计算时间

Tab. 1 Computation time of each layer /ms

任务单元	1	2	3	4	5	6
前向传播时间	0.886	0.086	0.542	1.293	0.172	0.007
反向传播时间	0.599	0.907	1.507	0.033	1.473	0.016

根据每一层的前向和反向的计算时间，可以划分工作段，结果如表 2 所示，其中 f_k 代表第 k 层的前向传播计算，b_k 代表第 k 层的反向传播计算，然后得到表 3 中的串行时间与并行时间比较情况：

表 2 工作段划分

Tab. 2 Division of working part /ms

工作段	分配情况	工作段时间
1	f1,f2,f3	1.514
2	f4,f5,f6,b6	1.488
3	b5,b4	1.506
4	b3	1.507
5	b2,b1	1.506

表 3 串行时间与并行时间比较

Tab. 3 Comparison of serial time and parallel time

计算类型	线程数	计算时间
串行计算	1	4035s
并行计算	5	1074s

由串行计算与并行计算的时间可以得出加速比为：Sp=4035/1074=3.757；设备利用率为：η=3.757/5=75.14%。可以看出，在使用了 4 个计算核的情况下，PipeCNN 可以取得良好的加速比以及设备利用率。

模型在测试数据集上的准确率是衡量模型好坏的重要指标，因此需要研究使用 PipeCNN 方法对模型准确率的影响。连续型 PipeCNN 在计算下一批次梯度的过程中，为了提高设备利用的效率而使用了异步梯度更新的方式，这可能会从某种程度上削减小批量梯度下降方法的效果。在实验中统计每个 Epoch 训练完成之后模型在测试数据集上的准确率，串行方式与 PipeCNN 方式的准确率结果对比如图 7 所示。

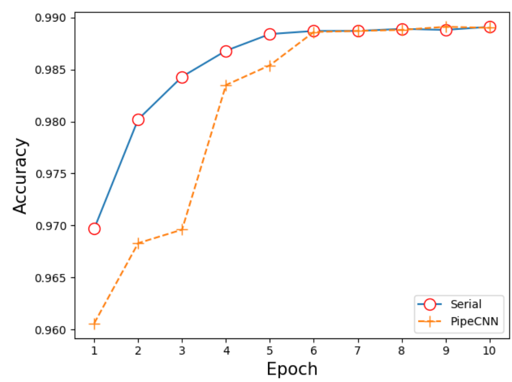


图 7 串行方式与 PipeCNN 的准确率比较

Fig. 7 Comparison of the accuracy between serial method and pipecnn

从图 7 中可以观察到，在训练的开始阶段，使用串行方式训练模型得到的准确率要高于 PipeCNN 的准确率，这是异步梯度更新带来的后果，但是随着训练过程的持续进行，PipeCNN 的准确率逐渐接近串行方式的准确率，说明 PipeCNN 方法对模型训练的后期影响较小。

训练完成后，取 10 个 Epoch 中准确率最高的模型为最终的结果，可得出使用串行方式与 PipeCNN 方式的模型准确率都为 98.91%。从实验结果可以看出，在 Epoch 设置较大的情况下，使用 PipeCNN 方法并不会影响到模型的训练精度，同时可以很好的加快卷积神经网络的训练速度。

4.2 Cifar10 数据集实验

其次进行训练识别彩色图像的实验，数据集采用 Cifar10，其中共有 10 个类别。不同于单通道的 MNIST 数据集，Cifar 数据集中的图片有三个通道，因此训练与识别的难度要比 MNIST 数据集大得多。实验中设置 Batch Size 为 16， Epoch 为 10，根据 Cifar10 数据集可设计出 14 层的卷积神经网络，表 4 所示为统计得到的训练过程中每一层的平均计算时间。

表 4 每一层的计算时间

Tab. 4 Computation time of each layer /ms

任务单元	1	2	3	4	5	6	7
前向传播时间	2.108	9.049	0.272	4.905	9.021	0.122	4.479
反向传播时间	3.042	15.695	0.312	10.931	16.404	0.164	13.409
任务单元	8	9	10	11	12	13	14
前向传播时间	11.056	9.456	8.933	0.076	0.407	0.022	0.008
反向传播时间	29.977	29.975	19.093	0.112	0.710	0.047	10.001

根据任务分配算法可以得到表 5 的工作段划分结果，然后计算出表 6 中的串行时间与并行时间比较情况：

表 5 工作段划分

Tab. 5 Division of working part /ms

工作段	任务单元	Total time
1	f1,f2,f3,f4,f5,f6,f7	29.956
2	f8,f9,f10,f11,f12,f13,f14	29.958
3	b14,b13,b12,b11,b10	29.963
4	b9	29.975
5	b8	29.977
6	b7,b6,b5	29.977
7	b4,b3,b2,b1	29.98

表 6 串行时间与并行时间比较

Tab. 6 Comparison of serial time and parallel time

计算类型	线程数	计算时间
串行计算	1	30243s
并行计算	7	5499s

训练完成后，在所有 Epoch 中取准确率最高的模型作为最终的训练结果，使用串行方式与 PipeCNN 的方式得到模型的最终准确率都为 60.84%。接下来衡量并行计算的效果，由串行计算与并行计算的时间可得加速比为：Sp=30243/5499=5.50；设备利用率为：η=5.50/7=78.6%。因此可以得出，PipeCNN 在彩色图像的训练识别任务中也可以获得良好的加速效果。

5 结束语

PipeCNN 采用软件流水线的思想来加速梯度计算的过程，消除了 GPipe 中不可避免的空闲阶段，同时也保证了模型训练的正确性，采用将前向传播与反向传播灵活分配在不同计算设备中的方法，理论上可以提高设备利用的效率。根据梯度更新方式的不同，PipeCNN 可分为间断与连续两种类型，其中连续型 PipeCNN 可更大程度减少流水线的空闲等待时间。在 MNIST 数据集和 Cifar10 数据集的实验中，PipeCNN 加速了卷积神经网络的训练过程，取得了良好的加速比以及设备利用率，这表明 PipeCNN 是一种通用的并行化卷积神经网络的方法。

下一步研究将进一步优化任务分配的过程，为了获得卷积神经网络中每一层的计算时间，需要先将程序运行一次，记录每一层的执行时间，再进行任务划分，而这种做法是冗余的，由于当网络定义完成的时候，每一层的计算量是可以

衡量的，因此在后续的研究中，可以依据模型本身的大小，估计每一层的计算量，从而可以直接得到每一层的计算量大小而不是根据统计每一层的计算时间来划分工作段。

参考文献：

[1] Khan A, Sohail A, Zahoora U, *et al.* A Survey of the Recent Architectures of Deep Convolutional Neural Networks [J]. ArXiv preprint arXiv: 1901. 06032 (2019) .

[2] He Kaiming, *et al.* Deep Residual Learning for Image Recognition [C]// IEEE Conference on Computer Vision and Pattern Recognition (CVPR) . 2016: 770-778.

[3] Tan Mingxing. Efficientnet: Rethinking model scaling for convolutional neural networks [J]. ArXiv preprint arXiv: 1905. 11946, 2019.

[4] Kirillov A, *et al.* PointRend: Image Segmentation as Rendering [J]. ArXiv preprint arXiv: 1912. 08193, 2019;

[5] Charles R, *et al.* Deep Hough Voting for 3D Object Detection in Point Clouds [J]. ArXiv preprint arXiv: 1904. 09664, 2019.

[6] He Kaiming, *et al.* Mask r-cnn [C]// Proceedings of the IEEE international conference on computer vision. 2017: 2961-2969.

[7] 沈夏炯，侯柏成，韩道军. 基于流水线的增强型植被指数快速提取算法 [J]. 计算机应用研究, 2018, 35 (09): 273-276. (Shen Xiajiong, Hou Bocheng, *et al.* Fast extraction algorithm of enhanced vegetation endex based on pipeline [J]. Application Research of Computers, 2018, 35 (09): 273-276.)

[8] Liu Bozhong, *et al.* Software pipelining for graphic processing unit acceleration [J]. International Journal of High Performance Computing Applications, 2015, 30 (2): 169-185.

[9] 刘家兵，徐云，等. X86 平台上 Open64 软件流水的设计与实现 [J]. 计算机工程, 2013, 39 (9): 16-18. (Liu Jiabing, Xu Yun, *et al.* Design and implementation of software pipelining for Open64 on X86 platform [J]. Computer Engineering, 2013, 39 (9): 16-18.)

[10] 陈纪孝，李勇. 软件流水循环缓冲的设计与实现 [J]. 计算机科学, 2013, 40 (4): 41-43. (Chen Jixiao, Li Yong. Design and implementation of software pipelined looped loop buffer [J]. Computer Science, 2013, 40 (4): 41-43.)

[11] 魏海涛，于俊清，余华飞，等. 一种面向数据流程序的软件流水并行化方法 [J]. 计算机学报, 2011, 34 (5): 131-140. (Wei Haitao, Yu Junqing, Yu Huafei *et al.* A method on software pipelined parallelism for data flow programs [J]. Chinese Journal Of Computers, 2011, 34 (5): 891-896.)

[12] Huang Yanping, *et al.* Gpipe: Efficient training of giant neural networks using pipeline parallelism [C]// Advances in Neural Information Processing Systems. 2019: 103-112.

[13] 舒嘉明，安虹，武铮，陈俊仕. 面向神威•太湖之光的一种通用并行卷积算法 [J]. 计算机工程, 2019, 45 (12): 153-159. (Shu Jiaming, An Hong, *et al.* A general parallel convolution algorithm for the Sunway Taihu Light [J]. Computer Engineering, 2019, 45 (12): 153-159.)

[14] Yang Jing, Yang Guanci. Modified Convolutional Neural Network Based on Dropout and the Stochastic Gradient Descent Optimizer [J]. Algorithms, 2018, 11 (3): 28-30.

[15] Feyzmahdavian H R, Aytekin A, Johansson M. An asynchronous mini-batch algorithm for regularized stochastic optimization [J]. IEEE Transactions on Automatic Control, 2016, 61 (12): 3740-3754.